Relazione sul progetto di risoluzione di sistemi lineari con metodi iterativi

-

Metodi del calcolo scientifico 2024/25

Ferrario, Christian
c.ferrario30@campus.unimib.it

Lanza, Andrea
a.lanza13@campus.unimib.it

June 5, 2025

Contents

1 Introduzione **3**
- 1.1 Contesto teorico . . . 3
- 1.2 Descrizione del progetto . . . 3
- 1.3 Scelte implementative . . . 4
  - 1.3.1 Supporto di matrici sparse . . . 4
  - 1.3.2 Criterio di arresto . . . 4
  - 1.3.3 Procedura di Validazione . . . 5
  - 1.3.4 Efficienza computazionale . . . 6
- 1.4 Gestione dei dati e visualizzazione . . . 6

2 Struttura della progetto **7**
- 2.1 Panoramica generale . . . 7
- 2.2 Libreria `linear_solver/` . . . 8
- 2.3 Programma eseguibile `demo/` . . . 8
- 2.4 Considerazioni . . . 9

3 Metodi Iterativi Implementati **10**
- 3.1 Metodo di Jacobi . . . 10
- 3.2 Metodo di Gauss-Seidel . . . 11
- 3.3 Metodo del Gradiente . . . 12
- 3.4 Metodo del Gradiente Coniugato . . . 13

4 Risultati sperimentali **15**
- 4.1 Analisi preliminari delle matrici . . . 15
- 4.2 Modalità di test . . . 15
- 4.3 Risultati dei test . . . 16
  - 4.3.1 Matrice spa1.mtx . . . 17
  - 4.3.2 Matrice spa2.mtx . . . 18
  - 4.3.3 Matrice vem1.mtx . . . 19
  - 4.3.4 Matrice vem2.mtx . . . 20

5 Sintesi e confronto dei metodi iterativi **21**

6 Considerazioni finali **21**

1 Introduzione

1.1 Contesto teorico

La risoluzione di sistemi lineari $Ax = b$ è un tema centrale nel calcolo scientifico. Sebbene esistano metodi **diretti** (come eliminazione di Gauss o fattorizzazione LU) che portano alla soluzione esatta in aritmetica reale, queste tecniche presentano alcune limitazioni, soprattutto in presenza di matrici mal condizionate o di grandi dimensioni, dove il fenomeno del *fill-in* può compromettere l'efficienza computazionale.

Per problemi su larga scala, si ricorre spesso a **metodi iterativi**, che costruiscono una successione di soluzioni approssimate convergenti alla soluzione esatta. Rispetto ai metodi diretti, essi offrono una maggiore flessibilità in termini di memoria e possono sfruttare la struttura sparsa della matrice, riducendo sensibilmente i costi computazionali.

I metodi iterativi si suddividono principalmente in due categorie:

- **Stazionari**, come Jacobi e Gauss-Seidel: semplici da implementare, ma sensibili al condizionamento della matrice;
- **Non stazionari**, come Metodo del Gradiente e Gradiente Coniugato: più complessi, ma generalmente più rapidi ed efficaci, in particolare quando la matrice è simmetrica e definita positiva.

La distinzione tra i due tipi risiede nel fatto che i metodi stazionari utilizzano una formula iterativa che rimane invariata a ogni passo, mentre i metodi non stazionari aggiornano dinamicamente i parametri di iterazione (come direzioni o coefficienti) in base al residuo corrente, migliorando spesso la velocità di convergenza.

1.2 Descrizione del progetto

Lo scopo del progetto è stato sviluppare una mini-libreria in Python per risolvere sistemi lineari mediante diversi metodi iterativi, **limitandosi al caso in cui la matrice A sia simmetrica e definita positiva**, con l'obiettivo finale di confrontare l'efficacia dei diversi metodi in termini di accuratezza, tempo di esecuzione e numero di iterazioni, applicandoli ad una serie di matrici sparse fornite in formato sparso .mtx e variando la tolleranza richiesta sulla soluzione.

Nel progetto sono stati implementati e confrontati quattro metodi:

1. **Metodo di Jacobi**

2. **Metodo di Gauss-Seidel**
3. **Metodo del Gradiente**
4. **Metodo del Gradiente Coniugato**

1.3 Scelte implementative

Il progetto è stato interamente sviluppato in **Python**, linguaggio scelto per la sua ampia diffusione in ambito scientifico e disponibilità di librerie numeriche avanzate.

1.3.1 Supporto di matrici sparse

In particolare, è stata utilizzata la libreria **SciPy**, in particolare il modulo *scipy.sparse*, al fine di garantire efficienza e compatibilità con **matrici sparse** in formato .mtx.
La libreria è stata impiegata esclusivamente per le operazioni di base su vettori e matrici (somma, prodotto matrice-vettore, norme), evitando l'utilizzo di qualsiasi funzione già implementata per la risoluzione diretta o iterativa di sistemi lineari, come richiesto dal progetto. Oltre ai metodi iterativi sopra indicati si é reso quindi necessario implementare anche il metodo risolutivo per **matrici triangolari inferiori**, utilizzato dal metodo di Gauss-Seidel, in modo che supportasse matrici in **formato sparso**.

1.3.2 Criterio di arresto

Tutti i metodi iterativi implementati partono da vettore iniziale nullo e si arrestano al soddisfacimento del criterio basato sulla norma relativa del residuo:

$$\frac{\|Ax^{(k)} - b\|}{\|b\|} < \text{tol}$$

dove tol è un parametro di **tolleranza** specificato dall'utente, oppure qualora venga superato un **numero massimo di iterazioni** maxIter, di default a 20.000 iterazioni, in modo da prevenire cicli infiniti in caso di mancata convergenza.

1.3.3 Procedura di Validazione

Per verificare il corretto funzionamento dei metodi iterativi implementati e valutarne le prestazioni in condizioni controllate, è stata seguita la procedura standard di validazione descritta nel testo dell'esercizio. Tale procedura prevede la generazione artificiale del termine noto b, partendo da una soluzione esatta nota, al fine di poter confrontare accuratamente la soluzione calcolata dai metodi iterativi con quella teorica.

La procedura si articola nei seguenti passaggi:

1. **Definizione della soluzione esatta:** viene fissato un vettore $x \in \mathbb{R}^n$ tale che
$$x = \begin{bmatrix} 1 & 1 & \dots & 1 \end{bmatrix}^T$$
ovvero un vettore colonna in cui tutte le entrate sono uguali a 1.

2. **Costruzione del termine noto:** dato il vettore x e una matrice A simmetrica definita positiva (le matrici `spa1.mtx`, `spa2.mtx`, `vem1.mtx`, `vem2.mtx`), si costruisce il termine noto come
$$b = Ax$$

3. **Risoluzione del sistema:** si risolve il sistema lineare
$$A\hat{x} = b$$
utilizzando ciascuno dei quattro metodi iterativi implementati (Jacobi, Gauss-Seidel, Gradiente, Gradiente Coniugato), partendo da una condizione iniziale nulla (cioè $\hat{x}^{(0)} = 0$).

4. **Valutazione dell'errore relativo:** si confronta la soluzione approssimata $\hat{x}$ ottenuta da ciascun metodo con la soluzione esatta x, calcolando l'errore relativo in norma euclidea:
$$\text{Errore relativo} = \frac{\|\hat{x} - x\|_2}{\|x\|_2}$$

5. **Ripetizione con diverse tolleranze:** la procedura è stata eseguita per diversi valori di tolleranza:
$$\text{tol} \in \left\{10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}\right\}$$
per valutare l'impatto della precisione richiesta sulle performance e sul numero di iterazioni necessarie.

Questa strategia consente di validare sia la correttezza dell'implementazione dei metodi, sia la loro capacità di convergere alla soluzione teorica con un livello di accuratezza controllabile. Inoltre, permette un confronto equo tra i metodi iterativi in termini di velocità di convergenza e stabilità numerica.

1.3.4 Efficienza computazionale

Un'attenzione particolare è stata posta anche sull'efficienza: si è scelto di **calcolare esplicitamente il residuo** $r = Ax - b$ ad ogni iterazione e usarlo direttamente per valutare il criterio di arresto, evitando di calcolare Ax due volte per ogni iterazione. Questo ha permesso di ottimizzare i tempi di esecuzione, pur mantenendo la piena correttezza numerica.

1.4 Gestione dei dati e visualizzazione

Le matrici fornite nel formato `.mtx` sono state caricate utilizzando le funzionalità di `scipy.io`, convertendole in formato `csr_matrix` per garantire operazioni efficienti su strutture sparse. I risultati ottenuti dai vari metodi sono stati raccolti e analizzati automaticamente attraverso moduli dedicati alla valutazione (`benchmark.py`) e alla generazione di grafici comparativi (`compare_plot.py`).

Tutti i test sono stati effettuati tramite uno script centralizzato, che esegue in sequenza i metodi implementati su tutte le matrici e tolleranze richieste, registrando numero di iterazioni, tempo di esecuzione e errore relativo.

Nei paragrafi successivi si descrivono l'architettura della libreria sviluppata, i dettagli implementativi dei metodi e i risultati ottenuti su matrici sparse reali.

2 Struttura della progetto

L'implementazione del progetto è stata organizzata in modo modulare, seguendo una separazione tra la **libreria** sviluppata e il **programma eseguibile** che ne fa uso. Di seguito si descrivono le principali componenti.

```
linear-iterative-solver
├── demo
│   ├── __init__.py
│   ├── cli.py
│   ├── constants.py
│   ├── logger.py
│   └── main.py
├── linear_solver
│   ├── analysis
│   │   ├── benchmark.py
│   │   ├── compare_plot.py
│   │   └── plot.py
│   ├── convergence
│   │   └── criteria.py
│   ├── matrix_analysis
│   │   ├── __init__.py
│   │   └── structure.py
│   └── solvers
│       ├── __init__.py
│       ├── base_solver.py
│       ├── conjugate_gradient.py
│       ├── gauss_seidel.py
│       ├── gradient.py
│       ├── jacobi.py
│       └── lower_triangular.py
├── matrices
│   ├── spa1.mtx
│   ├── spa2.mtx
│   ├── vem1.mtx
│   └── vem2.mtx
├── pyproject.toml
└── requirements.txt
```

2.1 Panoramica generale

La root della repository contiene tre directory principali:

- `linear_solver/`: libreria contenente l'implementazione dei metodi iterativi e delle funzioni di supporto
- `demo/`: script eseguibile e per il confronto sperimentale dei metodi

- `matrices/`: file `.mtx` contenenti le matrici sparse fornite per i test

Completano il progetto i file `pyproject.toml` e `requirements.txt`, che definiscono le dipendenze del progetto e script di esecuuzione.

2.2 Libreria `linear_solver/`

La libreria è suddivisa in diversi moduli:

- `solvers/`: contiene l'implementazione dei quattro metodi iterativi richiesti (`jacobi.py`, `gauss_seidel.py`, `gradient.py`, `conjugate_gradient.py`) oltre a `lower_triangular.py`, necessario per il metodo di Gauss-Seidel. Tutti i metodi derivano da una **classe astratta** comune definita in `base_solver.py`, che stabilisce l'*interfaccia standard* da implementare
- `convergence/criteria.py`: contiene i **criteri di arresto** utilizzati nei metodi iterativi, per il momento è implemetata esclusivamente la verifica sulla norma del residuo, ma l'utente può definire differenti criteri d'arresto, e passarli al metodo risolutivo durante la costruzione dell'oggetto
- `analysis/`: modulo dedicato alla **valutazione sperimentale**, con funzioni per la generazione di grafici e un file `benchmark.py` per la misurazione delle prestazioni dei metodi, in particolare del tempo di computazione, il numero di iterazioni e la soluzione finale, per poi calcolarne l'errore
- `matrix_analysis`: contiene funzioni utili per **verificare le proprietà delle matrici**, come la simmetria e la definitezza positiva

2.3 Programma eseguibile `demo/`

La directory `demo/` contiene gli script per eseguire la libreria sui dati di test:

- `main.py`: script principale che carica le matrici sparse dalla directory `matrices/`, genera il vettore $x = \mathbf{1}$, calcola $b = Ax$ e applica i quattro metodi iterativi per ciascun valore di tolleranza specificato;
- `cli.py`: interfaccia a riga di comando per la configurazione dei parametri;
- `logger.py` e `constants.py`: gestione dell'output e configurazioni globali.

2.4 Considerazioni

La struttura modulare del progetto garantisce una **buona separazione delle responsabilità**, facilita l'estensione (es. aggiunta di nuovi metodi) e favorisce la riusabilità della libreria in altri contesti. L'uso di un'architettura orientata agli oggetti per i solver migliora inoltre la leggibilità e la manutenibilità del codice.

3 Metodi Iterativi Implementati

In questa sezione vengono descritti i quattro metodi iterativi implementati nel progetto: Jacobi, Gauss-Seidel, Gradiente e Gradiente Coniugato. Per ciascun metodo, si fornisce una breve descrizione teorica e lo pseudocodice.

3.1 Metodo di Jacobi

Descrizione Teorica

Il metodo di Jacobi è un algoritmo iterativo per la risoluzione di sistemi lineari del tipo $Ax = b$, dove A è una matrice diagonale dominante. L'algoritmo si basa sulla decomposizione della matrice A come $A = D + R$, dove D è la parte diagonale e R il resto. L'iterazione è definita da:

$$x^{(k+1)} = x^{(k)} + D^{-1}(b - Ax^{(k)})$$

Proprietà

- Converge se la matrice è a **dominanza diagonale** (condizione sufficiente ma non necessaria).
- Costo per iterazione: $O(n^2)$.
- Residuo calcolabile direttamente.

Pseudocodice

```
Input: A, b, x^(0), tol, maxIter
Output: x^(k+1)

1.  k = 0
2.  while k < maxIter:
3.      x_new = x_old + D^{-1}(b - A * x_old)
4.      if ||x_new - x_old|| < tol:
5.          break
6.      x_old = x_new
7.      k = k + 1
8.  return x_new
```

3.2 Metodo di Gauss-Seidel

Descrizione Teorica

Il metodo di Gauss-Seidel è simile al metodo di Jacobi, ma utilizza le informazioni più recenti disponibili durante l'iterazione. La matrice A è decomposta come $A = L + U$, dove L è la parte inferiore (inclusa la diagonale) e U la parte superiore. L'iterazione è definita da:

$$x^{(k+1)} = x^{(k)} + L^{-1}r^{(k)}$$

dove

$$r^{(k)} = b - Ax^{(k)}$$

ma siccome calccolare l'inverso di L, si risolve invece il sistema tramite il metodo lower_triangular_solve

$$Ly = r^{(k)}$$

Proprietà

- Converge se la matrice è a **dominanza diagonale** (condizione sufficiente ma non necessaria).
- Richiede risoluzione di un sistema triangolare per iterazione, costo $O(n^2)$.
- Ogni iterazione più onerosa rispetto a Jacobi, ma spesso più efficace.

Pseudocodice

```
Input: A, b, x^(0), tol, maxIter
Output: x^(k+1)

1.  k = 0
2.    L := lower_triangular(A)
3.  while k < maxIter:
4.        r = b - A @ x_old
5.      x_new = x_old + lower_triangular_solve(L, r)
6.  return x
```

3.3 Metodo del Gradiente

Descrizione Teorica

Il metodo del gradiente, noto anche come discesa del gradiente, è un algoritmo iterativo per la risoluzione di sistemi lineari simmetrici e definiti positivi. L'algoritmo minimizza la funzione obiettivo:

$$f(x) = \frac{1}{2}x^T Ax - b^T x$$

L'iterazione è definita da:

$$x^{(k+1)} = x^{(k)} - \alpha^{(k)} \nabla f(x^{(k)})$$

dove $\nabla f(x^{(k)}) = Ax^{(k)} - b$ e $\alpha^{(k)}$ è il passo ottimale calcolato come:

$$\alpha^{(k)} = \frac{r^{(k)T} r^{(k)}}{r^{(k)T} A r^{(k)}}$$

Proprietà

- Richiede che A sia **simmetrica e definita positiva**.
- Interpretabile come discesa del gradiente su una funzione quadratica.
- È **non stazionario**, costo per iterazione $O(n^2)$.
- Sensibile al condizionamento della matrice.

Pseudocodice

```
Input: A, b, x^(0), tol, maxIter
Output: x^(k+1)

1.  r = b - A x
2.  k = 0
3.  while k < maxIter:
4.      alpha = (r^T r) / (r^T A r)
5.      x = x + alpha * r
6.      r = b - A x
7.      if ||r|| < tol:
8.          break
9.      k = k + 1
10. return x
```

3.4 Metodo del Gradiente Coniugato

Descrizione Teorica

Il metodo del gradiente coniugato è un algoritmo iterativo efficiente per la risoluzione di sistemi lineari simmetrici e definiti positivi. L'algoritmo genera una sequenza di direzioni coniugate rispetto a A e aggiorna la soluzione lungo queste direzioni. L'iterazione è definita da:

$$
\begin{aligned}
r^{(k)} &= b - Ax^{(k)} \\
p^{(k)} &= r^{(k)} + \beta^{(k-1)} p^{(k-1)} \\
\alpha^{(k)} &= \frac{r^{(k)T} r^{(k)}}{p^{(k)T} A p^{(k)}} \\
x^{(k+1)} &= x^{(k)} + \alpha^{(k)} p^{(k)} \\
r^{(k+1)} &= r^{(k)} - \alpha^{(k)} A p^{(k)} \\
\beta^{(k)} &= \frac{r^{(k+1)T} r^{(k+1)}}{r^{(k)T} r^{(k)}}
\end{aligned}
$$

Proprietà

- Miglioramento del gradiente classico, converge in **al più n iterazioni** per matrici $n \times n$.
- Vettori direzione coniugati rispetto ad A.
- Iterazione al costo $O(n^2)$, ma molto più veloce in pratica.
- Soluzione ottimale in sottospazi di dimensione crescente.

Pseudocodice

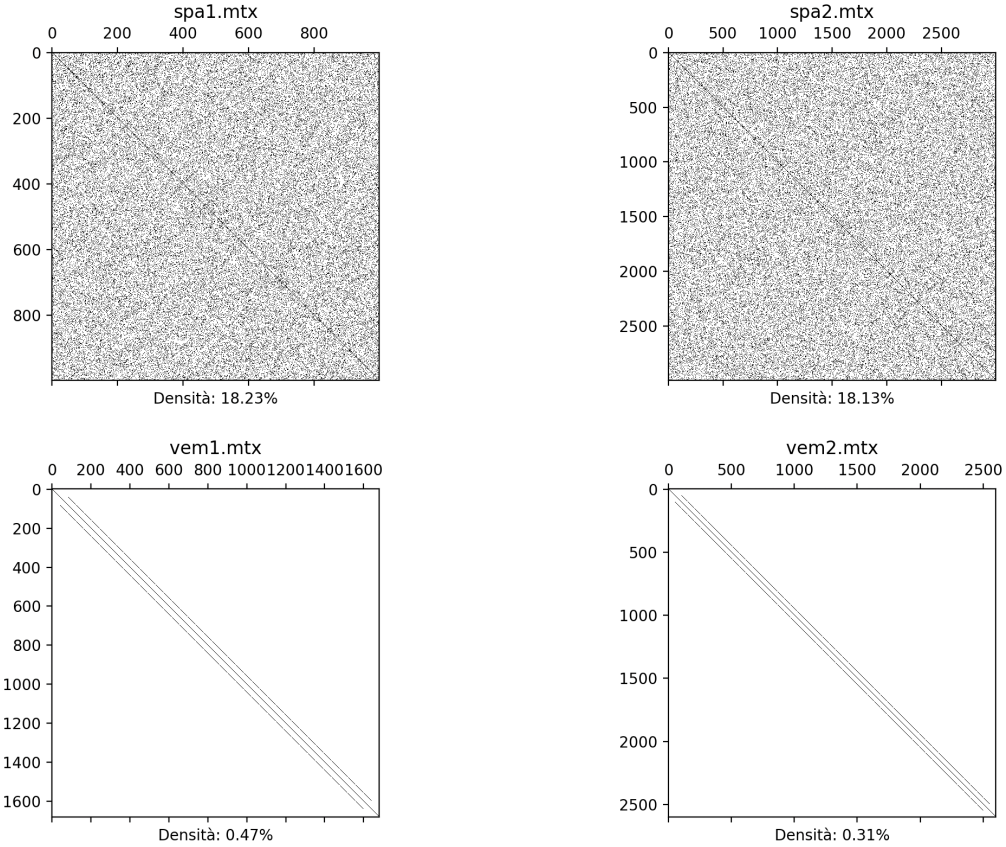
```
Input: A, b, x^(0), tol, maxIter
Output: x^(k+1)

1.  r = b - A x
2.  p = r
3.  k = 0
4.  while k < maxIter:
5.      alpha = (r^T r) / (p^T A p)
6.      x = x + alpha * p
7.      r_new = r - alpha * A p
8.      if ||r_new|| < tol:
9.          break
10.     beta = (r_new^T r_new) / (r^T r)
```

```
11.     p = r_new + beta * p
12.     r = r_new
13.     k = k + 1
14. return x
```

4 Risultati sperimentali

4.1 Analisi preliminari delle matrici

Le matrici utilizzate (`spa1.mtx`, `spa2.mtx`, `vem1.mtx`, `vem2.mtx`) sono ben note in letteratura scientifica e sono tratte da collezioni standardizzate (Matrix Market). Sono tutte **simmetriche e definite positive**, condizione che garantisce la **convergenza dei metodi iterativi**, in particolare per il metodo del gradiente coniugato.



4.2 Modalità di test

I metodi iterativi implementati sono stati testati secondo la procedura di validazione descritta in precedenza, al fine di verificarne la correttezza e confrontarne l'efficienza computazionale. In particolare, per ciascuna delle matrici fornite, è stato costruito un sistema lineare $Ax = b$ in cui la soluzione

esatta x è un vettore con tutte le componenti pari a 1 e il termine noto è stato calcolato come $b = Ax$.

Tutti i metodi sono stati eseguiti partendo dal vettore iniziale nullo e sono stati arrestati non appena è stata raggiunta la seguente condizione di convergenza:

$$\frac{\|Ax^{(k)} - b\|_2}{\|b\|_2} < \text{tol}$$

oppure, in alternativa, quando è stato raggiunto il numero massimo di iterazioni consentito, fissato a 20 000.

I test sono stati condotti per i seguenti valori di tolleranza:

$$\text{tol} \in \{10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}\}$$

Per ciascun esperimento sono stati registrati: l'errore relativo rispetto alla soluzione esatta, il numero di iterazioni effettuate e il tempo di esecuzione. I risultati ottenuti sono riportati nelle tabelle e nei grafici che seguono.

4.3 Risultati dei test

Tutti i metodi risolutivi convergono ad una soluzione per ciascuna delle matrici in esame, dato un numero massimo di iterazioni sufficentemente alto. Di seguito i risultati dettagliati suddivisi per matrice.

4.3.1 Matrice spa1.mtx

Table 1: Risultati per la matrice `spa1.mtx`

solver_class	tolerance	relative_error	execution_time	iterations
Conjugate	1e-10	1.20e-09	1.88e-02	200
GaussSeidelSolver	1e-10	1.21e-08	8.89e-01	32
GradientSolver	1e-10	9.81e-08	2.26e+00	12920
JacobiSolver	1e-10	1.73e-09	2.75e-02	314
Conjugate	1e-08	1.32e-07	1.66e-02	177
GaussSeidelSolver	1e-08	9.23e-07	7.09e-01	25
GradientSolver	1e-08	9.81e-06	1.44e+00	8234
JacobiSolver	1e-08	1.70e-07	2.18e-02	248
Conjugate	1e-06	2.55e-05	1.26e-02	134
GaussSeidelSolver	1e-06	7.02e-05	5.06e-01	18
GradientSolver	1e-06	9.67e-04	6.41e-01	3578
JacobiSolver	1e-06	1.68e-05	1.77e-02	182
Conjugate	1e-04	2.08e-02	5.13e-03	49
GaussSeidelSolver	1e-04	9.85e-03	2.67e-01	10
GradientSolver	1e-04	3.45e-02	2.85e-02	144
JacobiSolver	1e-04	1.65e-03	1.09e-02	116

I risultati mostrano chiaramente che il metodo del **Gradiente Coniugato** e **Jacobi** sono i metodi più efficienti in termini di tempo di esecuzione e numero di iterazioni per tutte le tolleranze testate, mantenendo sempre un errore relativo molto basso. Il **Gauss-Seidel** converge rapidamente con poche iterazioni, ma il tempo di esecuzione è due ordini di grandezza maggiore rispetto a Jacobi e Gradiente Coniugato. Il metodo del **Gradiente** richiede un numero molto elevato di iterazioni e tempi analoghi al metodo di Guss-Seidel, soprattutto con tolleranze stringenti.

4.3.2 Matrice spa2.mtx

Table 2: Risultati per la matrice `spa2.mtx`

solver_class	tolerance	relative_error	execution_time	iterations
Conjugate	1e-10	5.32e-09	6.24e-01	240
GaussSeidelSolver	1e-10	1.51e-09	3.44e+00	16
GradientSolver	1e-10	6.93e-08	4.29e+01	8286
JacobiSolver	1e-10	1.19e-09	2.59e-01	100
Conjugate	1e-08	5.59e-07	5.10e-01	196
GaussSeidelSolver	1e-08	7.57e-08	2.92e+00	13
GradientSolver	1e-08	6.85e-06	2.64e+01	5088
JacobiSolver	1e-08	1.26e-07	2.10e-01	79
Conjugate	1e-06	1.20e-04	3.27e-01	122
GaussSeidelSolver	1e-06	1.40e-05	1.93e+00	9
GradientSolver	1e-06	6.68e-04	1.04e+01	1950
JacobiSolver	1e-06	1.33e-05	1.53e-01	58
Conjugate	1e-04	9.82e-03	1.14e-01	42
GaussSeidelSolver	1e-04	7.13e-04	1.18e+00	6
GradientSolver	1e-04	1.80e-02	8.68e-01	162
JacobiSolver	1e-04	1.41e-03	9.95e-02	37

Anche per questa matrice il metodo del **Gradiente Coniugato** si conferma il più efficiente, con tempi ridotti e ottima accuratezza. Il **Gauss-Seidel** ha un numero di iterazioni molto basso, ma un tempo di esecuzione elevato, probabilmente dovuto all'alto costo per iterazione. Il metodo del **Gradiente** mostra ancora una volta tempi molto lunghi e molte iterazioni, rendendolo poco competitivo. **Jacobi**, pur richiedendo più iterazioni, ha un tempo di esecuzione contenuto e risultati affidabili.

4.3.3 Matrice vem1.mtx

Table 3: Risultati per la matrice `vem1.mtx`

solver_class	tolerance	relative_error	execution_time	iterations
Conjugate	1e-10	2.19e-11	2.63e-02	59
GaussSeidelSolver	1e-10	3.48e-09	6.57e+00	2339
GradientSolver	1e-10	2.69e-09	2.65e+00	3059
JacobiSolver	1e-10	3.52e-09	2.03e+00	4672
Conjugate	1e-08	2.83e-09	2.40e-02	53
GaussSeidelSolver	1e-08	3.49e-07	5.24e+00	1779
GradientSolver	1e-08	2.68e-07	2.03e+00	2337
JacobiSolver	1e-08	3.53e-07	1.53e+00	3553
Conjugate	1e-06	3.73e-07	2.02e-02	45
GaussSeidelSolver	1e-06	3.50e-05	3.36e+00	1219
GradientSolver	1e-06	2.69e-05	1.41e+00	1613
JacobiSolver	1e-06	3.53e-05	1.04e+00	2434
Conjugate	1e-04	4.08e-05	1.70e-02	38
GaussSeidelSolver	1e-04	3.48e-03	1.84e+00	660
GradientSolver	1e-04	2.68e-03	7.87e-01	891
JacobiSolver	1e-04	3.53e-03	5.90e-01	1315

Con matrici sparse meno strutturate, tutti i metodi iterativi faticano di più, ma il **Gradiente Coniugato** mantiene le migliori prestazioni. La differenza nei tempi di esecuzione tra i metodi aumenta: **Jacobi** diventa molto più lento, pur mantenendo un numero di iterazioni gestibile. **Gauss-Seidel** è penalizzato dal costo per iterazione. Il **Gradiente standard** risulta ancora il più lento insieme a Gauss-Seidel, con prestazioni globalmente inferiori.

4.3.4 Matrice vem2.mtx

Table 4: Risultati per la matrice `vem2.mtx`

solver_class	tolerance	relative_error	execution_time	iterations
Conjugate	1e-10	2.25e-11	7.99e-02	74
GaussSeidelSolver	1e-10	4.92e-09	1.81e+01	3590
GradientSolver	1e-10	3.78e-09	9.80e+00	4697
JacobiSolver	1e-10	4.95e-09	7.53e+00	7175
Conjugate	1e-08	4.30e-09	6.94e-02	66
GaussSeidelSolver	1e-08	4.93e-07	1.37e+01	2715
GradientSolver	1e-08	3.79e-07	7.43e+00	3567
JacobiSolver	1e-08	4.95e-07	5.86e+00	5426
Conjugate	1e-06	4.74e-07	6.08e-02	56
GaussSeidelSolver	1e-06	4.92e-05	9.31e+00	1841
GradientSolver	1e-06	3.77e-05	5.09e+00	2439
JacobiSolver	1e-06	4.95e-05	3.86e+00	3677
Conjugate	1e-04	5.73e-05	5.38e-02	47
GaussSeidelSolver	1e-04	4.93e-03	4.88e+00	966
GradientSolver	1e-04	3.79e-03	2.73e+00	1309
JacobiSolver	1e-04	4.96e-03	2.03e+00	1928

Per questa matrice, le differenze tra i metodi sono ancora più evidenti. Il metodo del **Gradiente Coniugato** risolve il sistema in meno di 100 iterazioni con tempi trascurabili rispetto agli altri. I metodi **Jacobi** e **Gauss-Seidel** necessitano di migliaia di iterazioni, diventando costosi anche in termini di tempo. Il metodo del **Gradiente** standard peggiora ulteriormente, sia in iterazioni che in tempo. Ciò conferma come l'efficienza algoritmica sia fortemente condizionata dalla struttura spettrale della matrice e dalla condizione del sistema.

5 Sintesi e confronto dei metodi iterativi

Dai risultati ottenuti su matrici di diversa struttura e dimensione, possiamo trarre alcune osservazioni generali sul comportamento dei metodi iterativi testati.

- **Conjugate Gradient (CG)**: si è dimostrato il metodo più efficiente nella maggior parte dei casi. Ha sempre richiesto il minor numero di iterazioni e il tempo di esecuzione più basso, pur garantendo un errore relativo contenuto anche con tolleranze molto strette. È particolarmente adatto per matrici simmetriche definite positive.
- **Gauss-Seidel**: ha mostrato una buona capacità di ridurre l'errore con poche iterazioni, ma i tempi di esecuzione sono risultati generalmente elevati, a causa del maggior costo computazionale per iterazione. In alcuni casi è risultato meno scalabile rispetto agli altri metodi.
- **Gradient Descent**: ha richiesto un numero di iterazioni molto elevato e tempi significativamente più lunghi, rendendolo il metodo meno efficiente del gruppo, specialmente per matrici di grandi dimensioni. Tuttavia, ha mostrato un comportamento stabile in termini di errore.
- **Jacobi**: ha presentato tempi di esecuzione contenuti, ma ha richiesto un numero di iterazioni medio-alto. Ha offerto una buona alternativa a Gauss-Seidel in termini di parallelizzabilità, pur con performance inferiori rispetto al metodo CG.

In generale, il metodo del **Gradiente Coniugato** rappresenta la scelta migliore in termini di efficienza e precisione per le matrici trattate. I metodi classici come **Jacobi** e **Gauss-Seidel** restano utili per casi didattici o in contesti in cui la semplicità implementativa è un fattore rilevante, mentre il **Gradiente semplice** è da evitare nei problemi su larga scala per la sua lentezza.

6 Considerazioni finali

I risultati ottenuti confermano le aspettative teoriche.

In termini di precisione, il comportamento dei metodi analizzati risente in modo significativo delle caratteristiche delle matrici utilizzate, in particolare del numero di condizionamento e della dimensione.

Per la matrice **spa1**, il metodo del gradiente semplice risulta il meno preciso, mentre gli altri tre metodi (Jacobi, Gauss-Seidel e gradiente coniugato) mostrano prestazioni simili. Questo è coerente con il fatto che il gradiente classico tende ad accumulare errori in presenza di matrici con condizionamento elevato (in questo caso ≈ 2000), riducendo la qualità dell'approssimazione anche se la matrice è simmetrica definita positiva (SPD).

Nel caso della matrice **spa2**, si osserva nuovamente un **comportamento meno preciso** del gradiente semplice, mentre Jacobi e Gauss-Seidel raggiungono risultati migliori. Questo può essere attribuito sia al condizionamento elevato (≈ 1400), sia alla dimensione molto ampia della matrice, che contribuisce ad aumentare la sensibilità numerica del gradiente. Al contrario, Jacobi e Gauss-Seidel, pur essendo metodi semplici, riescono in alcuni casi a produrre risultati più stabili quando la struttura della matrice favorisce una rapida riduzione dell'errore nei primi passi iterativi.

Diverso è il caso delle matrici **vem1** e **vem2**, caratterizzate da un **condizionamento più contenuto** ($\approx$ 350–500) e **dimensioni minori**. In questi casi, il metodo del gradiente coniugato mostra chiaramente le prestazioni migliori in termini di precisione, come previsto dalla teoria. Infatti, essendo progettato specificamente per matrici *SPD*, il gradiente coniugato sfrutta al meglio la struttura del problema, riuscendo a minimizzare l'errore in poche iterazioni e con alta stabilità numerica. Gli altri tre metodi si comportano in modo simile tra loro, ma senza raggiungere lo stesso livello di accuratezza.

In sintesi, il gradiente coniugato si conferma il metodo più preciso quando la matrice è ben condizionata, mentre il gradiente semplice mostra maggiori difficoltà su problemi mal condizionati o molto grandi. Jacobi e Gauss-Seidel, nonostante la loro semplicità, possono ottenere buoni risultati in contesti specifici, soprattutto quando il numero di iterazioni è limitato e l'errore non si accumula eccessivamente.